UNLIMITED

AD-A247 364

Report No. 91034

Report No. 91034

ROYAL SIGNALS AND RADAR ESTABLISHMENT, MALVERN

DTIC
ELECTE
MAR 1 3 1992
S D
D

THE VISTA STRUCTURED ASSEMBLER

Author: J Kershaw

This document has been approved for public release and sale; its distribution is unlimited.

PROCUREMENT EXECUTIVE, MINISTRY OF DEFENCE
RSRE
Malvern, Worcestershire.

January 1992

92 3 12 061

UNLIMITED

92-06594

CONDITIONS OF RELEASE

0120147 308889

********** DRIC U

COPYRIGHT (c)
1988
CONTROLLER
HMSO LONDON

********** DRIC Y

Reports quoted are not necessarily available to members of the public or to commercial organisations.

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Report 91034

TITLE: The VISTA Structured Assembler

AUTHOR: J Kershaw

DATE: January 1992

SUMMARY

VISTA is a structured assembly language for the VIPER microprocessor chip. Though the syntax of VISTA and the appearance of VISTA programs are reminiscent of a high-level language, the actual statements are VIPER machine instructions. VISTA provides the clarity and much of the convenience of a true high-level language without the need for a complex and possibly untrustworthy compiling program.

This Report describes the VISTA language by means of an annotated example, and gives instructions for using the VISTA translator. A formal grammar of VISTA, and descriptions of VIPER and the VIPER Object Program format, are included.

Accession For
NTIS CRARI
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

Copyright
©
Controller HMSO London
1992

INTENTIONALLY BLANK

Contents

INTENTIONALLY BLANK

1 Introduction

VISTA is a high-level assembly language: its statements are essentially VIPER machine instructions [Ker 87] embedded in a syntax resembling that of Algol 68. While it does not hide the structure of the underlying machine in the way that a true high-level language would, it does perform many of the routine housekeeping functions which can otherwise distract the programmer and make mistakes more likely.

The advantage of a relatively simple language like VISTA is that it does not need a complex compiler. Not only is the VISTA translator very much smaller and simpler than a typical compiler, but also its outputs are similar enough to the inputs to be checked easily. The outputs are printable, and annotated in such a way that mechanical checking is possible.

VISTA is not an inherently *safe* language, in the sense that some specialised languages [Cur 84] might be, though it does limit the programmer to structures which are reasonably free of complications. For example, there are no pointers or `GOTO` statements in VISTA. If VISTA is used to write safety-critical software, the program texts should be subjected to static analysis with a toolset such as MALPAS or SPADE. This process is made easy by the design of VISTA, which includes only constructs which can be analysed effectively by such toolsets. In fact the VISTA language was designed in parallel with a translator to the MALPAS input language [RTP], and many difficulties were resolved by changes to the language structure. A translator for SPADE [PVL] has also been developed.

This note describes VISTA in two ways: informally by means of an annotated example and formally with a modified BNF grammar. The example uses most of the constructs of the language, though one or two have escaped e.g. `INCLUDE`, `CONTINUE`. The grammar defines all possible constructs but does not in itself describe the semantics ("meaning") of a VISTA program; where this is not obvious it is described in the accompanying commentary.

The following program in VISTA is fairly trivial and makes no pretensions to safety: it reads in a text and then counts occurrences of selected words in that text. The line numbers are merely for reference in the notes below, and are not part of the program. Page layout of VISTA programs is completely free: the example follows normal conventions for indentation. Two pieces of hardware called **keyboard** and **screen** are assumed to be available for input and output.

The only aspects which will not seem reasonably familiar are the "region declarations" (lines 4 to 7) which define regions of memory into which code, constants, and so on are to be placed, and the fact that procedures in VISTA are declared *after* (rather than before) their calls. There is no profound reason for this: it simply seems more natural to put the main program first. No "loopholes" are allowed for recursion, which is forbidden in VISTA though a devious programmer with knowledge of the machine architecture can sneak it in and thereby probably lose the ability to analyse the program. Notice that `A`, `X`, `Y` are the VIPER machine registers, 32 bits each, and there is a single bit flag register called `B`. `X` and `Y` (only) can be used as indexes.

INTENTIONALLY BLANK

2 VISTA program example

```
 1  PROGRAM  word frequency counter.
 2
 3
 4  CODE  prog  FROM    0  TO  3999;          -- memory regions with bounds
 5  CONST const FROM  4000 TO  4095;
 6  DATA  data  FROM  16r8000  TO  16r8000+8191;
 7  PERI  peri  FROM    0  TO  1;
 8
 9  BEGIN
10  INT   last, count, match, k, p, q;        -- variables in DATA region
11
12  CHAN  keyboard, screen;                   -- in PERI region
13
14  INT   key[20], text[9000];                -- more data, vectors
15
16  INT   sp = 32, eof = 26, cr = 13;         -- in the CONST region
17  INT   ms1[ ] = "\nInput the text: \0\";         -- messages
18  INT   ms2[ ] = "\nKeyword: \0\";
19  INT   ms3[ ] = "\nNumber of occurrences is \0\";
20  INT   ms4[ ] = "outside range\0\";
21
22  -- End of variable and constant declarations, now for the program
23
24  A := 1;   CALL message;                   -- invite the user to type ..
25  A := 1;   last := A;                      -- .. in a sample of text
26  A := sp;  text[0] := A;
27
28  WHILE  (CALL getchar; CALL upper)  A /= eof
29  DO  X := last;                            -- in case getchar changes X
30      CASE  A >= 'A'  AND  A <= 'Z':  SKIP;       -- letters
31            A >= '0'  AND  A <= '9':  SKIP        -- digits
32      ELSE  A := sp                               -- all others
33      ESAC;
34      text[X] := A;  X := X + 1;            -- store the character
35      last := X
36  OD;
37
38  X := last;  A := sp;  text[X] := A;       -- terminator for last word
39
40  REPEAT  A := 2;  CALL message;               DISPLAY get new key word
41          A := 0;  p := A;
42
```

```
43          WHILE  (CALL getchar; CALL upper)  A /= cr
44          DO  X := p;
45              key[X] := A;                -- read & store key word
46              X := X + 1;
47              p := X
48          OD;
49
50          X := p;  A := sp;  key[X] := A;  -- terminator for key word
51          IF  X = 0  THEN  BREAK  FI;      -- exit if null key word
52          A := 0;  p := A;  count := A;
53
54          WHILE  (Y := 0; match := Y; X := p)  X < last
55          DO  WHILE  (A := key[Y])  A = text[X]
56              DO  IF  A = sp
57                  THEN  A := 1;
58                        match := A;       -- end of word & no mismatch
59                        BREAK
60                  FI;
61                  X := X + 1;
62                  Y := Y + 1              -- more letters, keep trying
63              OD;
64              IF  (A := match; X := p; Y := text[X-1])
65                   A /= 0  AND  Y = sp
66              THEN  A := count;           -- whole of key matched
67                    A := A + 1;           -- check space before word ..
68                    count := A            -- .. don't want to find ..
69              FI;                         -- e.g. "king" in "smoking"
70              X := X + 1;  p := X
71          OD;
72
73          A := 3;  CALL message;          -- print number of matches
74          A := count;  CALL print
75
76  UNTIL  (A := key[0])  A = sp;           -- until null key word input
77  STOP;
78
79  -- Now for the procedures, notice they follow the main program
80
81  PROC  getchar:  (WHILE  (A := INPUT keyboard)  A = 0  DO  SKIP  OD);
82
83
84  PROC  upper:  (IF A >= 'a' AND A <= 'z' THEN A := A-('a'-'A') FI);
85
```

```
 86
 87  PROC  print:                                -- unsigned decimal output
 88    BEGIN  INT  lzflag;
 89      X := 0;                                 -- suppress leading zeros
 90      lzflag := X;
 91      IF  A < 0  OR  A > 9999  THEN  A := 4;  CALL message
 92      ELSE  X := 1000;  CALL digit;
 93            X := 100;   CALL digit;
 94            X := 10;    CALL digit;         -- 4 digits maximum
 95            X := 1;     CALL digit
 96      FI;
 97
 98      PROC  digit:                -- prints 1 digit of number in A
 99        BEGIN  INT  power;        -- X must hold a power of 10, X /= 0
100          power := X;
101          Y := 0;
102          WHILE  A >= power  DO  A := A - power;  Y := Y + 1  OD;
103          IF  X = 1  OR  Y > 0  THEN  lzflag := X  FI;
104          Y := Y + '0';                      -- convert to ASCII
105          X := lzflag;
106          IF  X = 0  THEN  Y := sp  FI;      -- leading zero
107          OUTPUT Y, screen
108        END
109    END;     -- of "print"
110
111
112  PROC  message:                             -- outputs message selected ..
113    BEGIN  INT  select;                      -- .. by A register, 1 to 4
114      select := A;
115      X := 0;
116      WHILE  TRUE                            -- i.e. do forever, exit ..
117      DO  CASE  (A := select) A IN           -- .. by BREAK or RETURN
118                1:  (A := ms1[X]);
119                2:  (A := ms2[X]);           -- get 4 bytes from the message
120                3:  (A := ms3[X]);
121                4:  (A := ms4[X])
122          ELSE   A := 0
123          ESAC;
124
125          Y := A AND 255;                    -- now output the characters ..
126          IF  Y = 0  THEN  RETURN  FI;       -- .. terminating on a NULL
127          OUTPUT Y, screen;
128          CALL right8;
129          IF  Y = 0  THEN  RETURN  FI;       -- return from "message"
130          OUTPUT Y, screen;
131          CALL right8;
```

```
132          IF  Y = 0  THEN  RETURN  FI;
133          OUTPUT Y, screen;
134          CALL right8;
135          IF  Y = 0  THEN  RETURN  FI;
136          OUTPUT Y, screen;
137          X := X + 1
138       OD;
139
140       PROC  right8:                          -- 8 place right shift
141         BEGIN
142           A := A/2;  A := A/2;  A := A/2;  A := A/2;
143           A := A/2;  A := A/2;  A := A/2;  A := A/2;
144           Y := A AND 255
145         END
146
147     END       -- of "message"
148  END          -- of program
149  FINISH
```

3 Notes on the example (by line number)

1

All programs begin with **PROGRAM**, followed by an identifier which is taken as a title. The rest of the title line is ignored as comment. Identifiers in VISTA begin with a lower case letter and are made up of lower case letters and/or digits. They can be of any length though the translator will remember only the first 12 characters. The whole of a VISTA program must be processed at once - there is no "separate compilation" facility.

4..7

Region declarations. All storage in a VISTA program is allocated in a defined region, in ascending order of address. Constants are kept separate from data to allow use of ROM. The translator will report an error if a region fills up, or if region declarations overlap. The **PERI** region has a separate address space from the other three. Notice the use of decimal and hex numbers (the other bases allowed are **2r** for binary and **8r** for octal) and of constant expressions which can be arbitrarily complex.

All the region declarations must be at the start, before the first block opens.

9

BEGIN marks the opening of the main block. All the data and procedure declarations of

the program must be within this or within procedures enclosed by it. Blocks may not be nested, though procedures can be: compound statements enclosed by BEGIN .. END or round brackets (which are everywhere interchangeable) are allowed but they may not contain declarations. Local redeclarations of variables within procedures take precedence over less local declarations with the same names, in the usual way. Local redeclarations of scalar *constants* are required to have the same value as any less local declarations which may still be in scope.

10

Ordinary variable declarations in the DATA region. The only data types allowed are INT and BITS, and vectors thereof. Both types occupy one 32 bit word per element. There is no restriction on the use of either. All storage allocation in VISTA is static. The initial values of variables are undefined; in a typical VIPER system they will be the contents of a RAM chip which has just been switched on.

12

Peripheral addresses in the PERI region, usable only with INPUT or OUTPUT (see lines 81, 107).

14

Vectors of variables. The lower bound is fixed at 0. Only one dimension is allowed.

16

Constants, to be placed in the CONST region. These are just ASCII character values. Wherever VISTA expects a constant (e.g. here or as a vector size, line 14) a constant expression can be written using any of the usual arithmetic and logical operators. Once declared, a named constant (e.g. "eof") can be used in constant expressions. Constants declared as of type BITS are treated as unsigned in constant expressions, as are numbers denoted in hex, octal, or binary. Decimal numbers and named constants of type INT are treated as signed. The choice of operator (signed vs. unsigned) in a constant expression is based on the type of the *left* operand: the distinction matters in practice only for the operators >>, /, % (see below).

The operators permitted in constant expressions, in descending order of priority are:

- unary + - NOT applied to *primaries* i.e. single names or numbers, or bracketed expressions.
- << >> left/right shift the left operand by the number of places given by the right operand. Signed right shift fills with copies of the sign bit, unsigned with zeroes.

- `&` `|` bitwise logical AND, OR.
- `*` `/` `%` multiply, divide, modulo, defined as on the host machine.
- `+` `-` add, subtract.

Constants (scalar or vector) cannot be assigned to. Their values are fixed irrevocably by the declaration, as they would be in reality if stored in ROM.

17..20

Vectors of constants. These can be given values either as a list of constant expressions e.g. `(1, 2, 3, 4)` or (as here) a string of 7 bit ASCII characters. Characters are stored four per word, least significant first. Notice the special characters \n (line feed) and \0\ (character of value 0, used here as a string terminator). Characters can also be used as individual constants: the declaration on line 16 could equally well be written as

```
INT  sp = ' ',  eof = '\26\',  cr = '\c';
```

Character constants generate the standard ASCII representation without parity. See Section 5 ("initlist") for a list of special characters.

22

Comments follow `--` as in Ada, and extend to the end of the line.

24 et seq.

Instructions are translated one-for-one to VIPER machine instructions. There are three 32 bit registers called `A`, `X`, `Y` of which `X` and `Y` can be used as index registers. `Y` is changed by the `CALL` instruction, but otherwise the registers can be used interchangeably.

No explicit arguments are allowed with procedure calls, but `A` and `X` can be used freely to hand over parameters.

28

Notice the "preface block" `(CALL getchar; CALL upper)` which is optional before a condition to set up the register contents. Conditions generally compare a register with a variable or constant (*not* two registers or two variables) but there are also conditions `B`, `NOT B`, `TRUE`, `FALSE` for special cases.

30..33

This form of the VISTA **CASE** statement requires an explicit predicate on each limb. A single preface block is allowed immediately after the word **CASE**. Conditions are tested in the order written and never cause a change in the register values; when a condition is found to be true the corresponding limb is entered and followed by exit from the **CASE** statement. See comment on line 117 for the other form of **CASE** statement.

Notice the use of **AND** as a connective in conditionals. Any number of **AND** or **OR** connectives may be used, but not both in the same condition. No brackets are allowed. Without these restrictions (which arise because of the need to map VISTA statements into single VIPER instructions) it would have been possible to use a simple **IF - THEN - ELSE** here.

The **SKIP** statement has no effect (and generates no code) but is provided for clarity. It can be used or omitted as preferred.

32

The **ELSE** limb is optional. If it were omitted and no condition was true, a **CASE** statement would cause the machine to stop. If you want a **CASE** statement to do nothing in this situation, use **ELSE SKIP**.

40

Loop constructs are **WHILE - DO - OD** and **REPEAT - UNTIL** .

DISPLAY is an alternative form of comment which is preserved in the translator output file. It does not form part of the program but can be picked up by other software e.g. the VIPER simulator.

51

BREAK and **CONTINUE** are valid only in a **WHILE** or **REPEAT** statement. Their effect is, respectively:

- Exit from the most local loop, i.e. jump to the statement following **OD** or following the condition after **UNTIL**.
- Proceed to the next cycle of the most local loop, i.e. jump to the beginning of the condition after **WHILE** or **UNTIL**.

The jump in this case is to line 77

64..65

This is a fairly elaborate condition! The whole thing compiles to 6 instructions.

77

The STOP statement generates an illegal VIPER instruction which stops the processor. A constant expression can be placed after the STOP, in which case its value will be compiled into the (otherwise unused) address part of the instruction and will be visible on the address bus when the processor has stopped. If no expression is given, the translator will insert the address of the instruction.

81

Procedure declarations in VISTA follow their calls, i.e. only forward reference is allowed. This is logically more satisfactory (the main program comes first) and matches MALPAS better than the traditional rule of declaration before use. Recursion is not in principle possible, since the language does not allow pointers of any kind.

Variables and constants (lines 10..20) must be declared before use in the conventional way. In this and most other respects VISTA conforms to the normal Algol or Pascal block structure. Round brackets are interchangeable with BEGIN .. END .

The peripheral location keyboard is assumed to deliver 0 until a character is typed.

84

upper turns lower case letters in A to upper case.

87

4 digit unsigned decimal output.

98

digit is nested within print .

102

Division by repeated subtraction, since VIPER has (at present) no divide instruction. The loop is never obeyed more than 9 times.

107

The peripheral location screen is assumed to make characters visible in some way.

116

WHILE TRUE gives an endless loop which can be left only by **BREAK** or (in a procedure) **RETURN**.

117..123

The strings output by **message** must be known to the procedure in advance. It is not possible to hand over the address of an arbitrary string since VISTA does not allow pointers. This is an example of the other form of **CASE** statement in which each limb has either a single selector value or a bounded range of values e.g. `1..9:` If the contents of the nominated register (**A** in this case) match the selector, the limb is entered. Selector values and bounds may be constant expressions, as usual. A minor variation replaces **IN** by **UNSIGNED**, telling the translator to use unsigned tests when checking whether the register is within a range.

This form of **CASE** is much easier for programs like MALPAS to analyse, e.g. when checking for coverage and overlap of the conditions.

126

A zero character terminates the string. Strings are stored with the least-significant 8 bits of the word holding the first character. **RETURN** exits from the most local procedure only.

140

`right8` is nested inside **message**.

142

These are "arithmetic" shifts which duplicate the sign bit. Line 144 renders this unimportant. The alternative is an end-around shift through **B** (e.g. `A >> 1`) which in this case would do just as well.

149

Every program ends nominally with **FINISH** though end-of-file alone is acceptable.

4 Constructs not included in the example

- **B := TRUE; B := FALSE**

  Direct setting/clearing of the **B** flag.

- **INCLUDE \path\filename.vis**

  Source text inclusion facility. The first non-printing character e.g. space or end-of-line, terminates the file name. There is no default extension. The example given is of an MSDOS file name; the syntax is that appropriate to the host machine on which the VISTA translator is running. INCLUDE files may be nested up to 9 deep. A list of the files used will be printed at the top of the diagnostic output file.

5 Commentary on the grammar of VISTA

A modified form of the BNF grammar which follows is input to the SID parser generator, whose output forms the kernel of the translator. This version of the grammar has been shorn of the "compiling actions" which form the interface to the rest of the translator, but do not change the structure of the language except in the handling of character strings where the effect of an absent action has been embodied into the syntax for clarity.

Terminal symbols e.g. NUM, SEMI, BEGIN are in capitals, non-terminals in lower case. Comments are between \ .. \ . Punctuation characters (for instance = , ;) are part of the meta-language of this version of BNF: commas separate alternatives and the last alternative of a rule ends with a semicolon. Empty alternatives are indicated by void.

The commentary is indexed by the names of the non-terminals, i.e. those which appear to the left of an = sign.

prog

The leading rule. Every program begins with PROGRAM and a name; any text following on the same line is taken as comment.

regions

A "region" declaration defines a bounded region of store or peripheral addressing space into which objects of appropriate type (CODE, CONST, DATA or PERI) can be placed by the compiler. Only objects in DATA regions can be assigned to; only objects in PERI regions can be used in INPUT or OUTPUT instructions. Regions are global to the whole program, and the space they occupy is allocated permanently. The address space for PERI regions is separate from the other three. Region bounds are inclusive.

Several region declarations of the same type may be made e.g.

```
CODE  rom1  FROM   0   TO  4095;
CODE  rom2  FROM 65536 TO  65536 + 4095;
```

Only one region of each type (CODE, CONST, DATA, PERI) is in use at any one time, but it can be changed between declarations (for CODE, between procedures) by writing e.g.

CODE IN rom1; and reset to the region previously in use by e.g. RESET CODE; Changes are stacked separately for each type e.g.

```
CODE IN rom1;  DATA IN ram;  PROC  fun1:  (etc);
CODE IN rom2;  PROC  fun2:  (etc);
RESET CODE;
RESET CODE;    -- restore status quo
RESET DATA;
```

A region declaration automatically sets the new region as the current one of that type. While a region is in use, objects of appropriate type are placed at successively increasing addresses within that region until it is full, when an error message is given. While a region is not in use (because the programmer has changed to another of the same type) its current loading position is remembered.

All the region declarations in a program must be grouped together at the top, before the main block begins.

block

VISTA is block structured, where a block is either the main program or the body of a procedure. Blocks cannot be "nested" in VISTA, except as the bodies of procedures: BEGIN .. END or round brackets can be used to group instructions (e.g. for a CASE limb where the syntax demands a single instruction) but such a "compound statement" cannot include declarations. The present VISTA translator does not re-use memory allocated to local variables which cannot be in scope simultaneously, but this is not guaranteed to persist in future versions.

The scope rules are as follows:

- A "region" declaration is in scope for the whole program.
- An "object" (variable or constant) declaration is in scope from the point of declaration until the end of the main block or of the enclosing procedure, whichever is smaller.
- A "procedure" declaration is in scope from the start of the main block or of the enclosing procedure, whichever is smaller, until the start of its own declaration.

The sequence within a block is strictly defined, and is the same whether the block is the main program or a procedure body.

1. Variable and/or constant declarations (if any).
2. Main code, which can be void e.g in a program consisting entirely of procedures.
3. Procedure declarations which can be interspersed with more variable or constant declarations (e.g. those used only to communicate between procedures) or with region changes.

BEGIN .. END and **( )** are interchangeable but must be correctly paired.

procdec

No explicit arguments or return values are allowed. What you do with the registers is up to you, but remember that the **CALL** instruction changes **Y**. **RETURN** changes no register except the program counter which is not visible in VISTA.

The form **PROC ID FROM unsexp** declares that a procedure can be found at the absolute address given by the unsigned expression, and is used to call facilities in other (separately compiled) programs e.g. in a ROM-resident monitor. The monitor program would contain a **CODE** region declaration starting **FROM** the same address.

objdec

An "object" declaration allocates space in one or other of the current regions. Types **BITS** and **INT** allocate space in the **DATA** region, or in the **CONST** region if the object is initialised. Type **CHAN** allocates space in the **PERI** region.

Wherever a number would be appropriate in VISTA, a constant expression can be written. Objects declared as scalar constants e.g.

```
INT  maxrom = 4095;
```

can be used in constant expressions - their values are held in the compiler as well as (possibly) in the program. Scalar constants which will fit into 20 bits are usually embedded in literal instructions and not stored separately. Vectors of constants (or elements thereof) e.g.

```
BITS  table[ ] = (1, 2, 4, 8, 16, 32, 64, 128);
```

cannot be used in constant expressions. They are indexed from 0 up.

Vectors of variables e.g. **INT vec[100];** must be declared with the size (in 32 bit words) and are also indexed from 0 up, in this case to 99.

initlist

A vector of constants can be initialised to a string, which will be stored 4 ASCII bytes per word without parity, least significant byte first. There is no automatic terminator, though unfilled words will be padded with nulls. Certain special characters can be included in strings using "backslash" and a lower case letter e.g. **\n, \c, \f, \s, \t, \", ** for respectively linefeed, carriage return, form feed, space, tab, quote (which would otherwise terminate the string) and backslash. An arbitrary 8 bit constant can be placed in a string

(e.g. as a terminator) thus: `"A null-terminated string\0\"`. Any of the usual number bases may be used.

Strings must normally fit within one line. To extend a string on to the following line(s), end each line with the sequence `\e` thus:

```
INT message[ ] = "An example of a string which will not fit \e
comfortably within a single line";
```

`\e` does *not* insert a new line into the string: an explicit `\n` (or `\c\n`) must be used.

exp

Expressions are either all constant or consist of a variable only. Variables appear only in instructions. Unary minus or `NOT` are allowed only before a *primary* e.g. `-(const-1)` is allowed but `-const-1` is not. See the commentary on the example program for a list of permitted operators. Any identifiers used in an expression must be already declared and in scope, see above under "block".

inst

The syntax of VIPER instructions - most of the rest of VISTA is not VIPER-specific. Void is allowed as an instruction, but `SKIP` is provided for clarity e.g. as a null branch in `CASE` statements. It generates no code.

regassign, indexreg

Data registers (`sreg` or `dreg`) are `A, X, Y`. Index registers are `X, Y` only.

loopst

`BREAK, CONTINUE` are valid only in `WHILE` or `REPEAT` statements, and transfer control to the instruction following the innermost loop or the start of the controlling test of the innermost loop respectively.

casest

`CASE` statements come in two forms, one with an explicit predicate on each limb (`caseitem`) and one with a nominated register and a selection value or range on each limb (`setitem`). Either form can have an optional `ELSE` limb at the end, but if this is not present the translator will insert `ELSE STOP`. In both forms, the cases are tested in the order written. There is no check for coverage or overlap between predicates.

condlist

Conditions all correspond to single VIPER instructions, which compare a register with an immediate or stored operand. To set up the registers ready for the comparison, a "preface" can be inserted thus:

```
IF  (A := fred)  A > 0  THEN ..... FI;
WHILE (X := index; A := table[X])  A /= 0  DO .... OD;
```

The preface (if present) is part of the test - see **CONTINUE** above.

There is no particular advantage in doing **IF** statements this way: compare

```
A := fred;  IF A > 0 ...
```

but the same is not true for **WHILE** and **REPEAT** statements: consider

```
A := fred;  WHILE A > 0 DO .... ; A := 1; joe := A OD;
```

condition

Compound *conditions can be written using* ***AND*** *or* ***OR*** *but not both, because of* the restriction that each comparison is a single VIPER instruction. Every comparison is performed every time. The only possible side effect is a hardware error on address-out-of-bounds e.g. in

```
X := 5000000;  IF A = 0 AND Y >= table[X] THEN ... FI;
```

IF cond1 OR cond2 OR ... THEN is compiled as (see VIPER definition below)

```
TEST cond1
SET B IF cond2
SET B IF ...
IF NOT B GOTO else-or-fi
then-code
```

IF cond1 AND cond2 AND ... THEN is compiled as

```
TEST inverse of cond1
SET B IF inverse of cond2
SET B IF ...
IF B GOTO else-or-fi
then-code
```

"Condition" includes **TRUE** and **FALSE** for situations like **WHILE TRUE DO ... OD** and **B, NOT B** for special occasions.

relop

GE, LT are unsigned 32 bit comparisons.

operand, unsop

Things on the right-hand side of a **regassign** i.e. constant expressions, variables, vector references. If a constant expression can be fitted into 20 bits it will be handled as a literal, otherwise storage will be allocated for it in the current **CONST** region.

offset

A constant offset done at compile time, with an optional run-time index in the **X** or **Y** register.

variable

Writes to memory e.g. **fred := A** , also arguments of **INPUT** or **OUTPUT**.

function

These are *machine functions*, not to be confused with operators in constant expressions even though **+ -** can be used in either context. They appear only after a register name (**A, X** or **Y**).

shift

/2 >>1 *2 <<1 are the only permitted combinations in instructions, corresponding to the VIPER machine functions. The right operand is unrestricted in constant expressions.

INTENTIONALLY BLANK

6 VISTA Grammar

```
\********************  Basic language structure  **********************\

'RULES'

prog     =     PROGRAM  ID  regions  block  opsemi  FINISH;


regions  =     rdec  SEMI,                  \ All region decs are global \
               rdec  SEMI  regions;         \ .. and there must be >= 1 \


rdec     =     AREA  bounds,                \ data region declaration \
                                            \ AREA = CONST, DATA, or PERI \
               CODE  bounds,                \ CODE region declaration \
               datachange,                  \ change current region \
               codechange;


bounds   =     ID  FROM  unsexp  TO  unsexp  ;


block    =     BEGIN  decpart  codepart  END,
               OPEN   decpart  codepart  CLOSE;


decpart  =     void,                        \ a void decpart is allowed \
               ddec  SEMI  decpart;         \ note CODE change not allowed \


codepart =     inst,                        \ notice this can also be void \
               inst  SEMI  procode;


procode  =     codepart,
               procetc  proclist;


procetc  =     procdec,
               ddec,                        \ interspersed data decs .. \
               codechange;                  \ .. or region changes .. \
                                            \ .. but only between PROCs \
```

```
procdec     =   PROC  ID  COLON  block,
                PROC  ID  FROM  unsexp;

proclist    =   void,
                SEMI,                          \ have mercy on spare semis! \
                SEMI  procetc  proclist;

\******************  Data and channel declarations  **********************\

ddec        =   objdec,
                datachange;

objdec      =   TYPE  obj  objlist;            \ data or CHAN dec \
                                               \ TYPE = BITS, INT, or CHAN \

datachange  =   AREA  IN  ID,                  \ change data region \
                RESET  AREA;                   \ back to old region \

codechange  =   CODE  IN  ID,                  \ change CODE region \
                RESET  CODE;

obj         =   ID,
                ID  EQUALS  exp,               \ not with CHAN! \
                ID  OSQ  unsexp  CSQ,
                ID  OSQ  CSQ  EQUALS  initlist;

objlist     =   void,
                COMMA  obj  objlist;

initlist    =   OPEN  exp  exlist  CLOSE,
                QUOTE <string> QUOTE;          \ character string, see above \

exlist      =   void,
                COMMA  exp  exlist;
```

```
\   Constant expressions, including scalar identifiers declared as CONST  \
\    A scalar variable as an operand is included for syntactic reasons    \


exp         =   unsexp,
                PLUS  primary,              \ +ID or NOT ID still ..\
                NOT  primary,               \  qualifies as a variable ..\
                MINUS  primary;             \  while -ID does not \


unsexp      =   term,                       \ operators   +  -  \
                unsexp  PLUS  term,
                unsexp  MINUS  term;


term        =   factor,                     \ operators   *  /  %  \
                term  TIMES  factor,
                term  SLASH  factor,
                term  MOD  factor;


factor      =   shiftor,                    \ operators   &  |  \
                factor  AMPERSAND  shiftor,
                factor  BAR  shiftor;


shiftor     =   primary,                    \ operators  >>  <<  \
                shiftor  GREATER  GREATER  primary,
                shiftor  LESS  LESS  primary;


primary     =   ID,
                NUM,
                BIGNUM,                     \ over-size number \
                SIZE  ID,
                OPEN  unsexp  CLOSE;
```

```
\ Executable statements.  This includes nearly all the VIPER-specific bits  \

inst        =   void,                       \ to allow spare semicolons \
                simstat,                    \ those allowed in prefaces \
                SKIP,                       \ explicit null instruction \
                RETURN,
                STOP,
                STOP  exp,                  \ qualifier for diagnostics \
                loopst,
                BEGIN  codelist  END,       \ compound statements \
                OPEN   codelist  END;


simstat     =   regassign,
                variable  COLON  EQUALS  sreg,
                CALL  ID,
                OUTPUT  sreg  COMMA  variable,
                IF  condlist  THEN  codelist  elsepart  FI,
                casest;


regassign   =   dreg  COLON  EQUALS  sreg ,
                dreg  COLON  EQUALS  NOT sreg,
                dreg  COLON  EQUALS  operand,
                dreg  COLON  EQUALS  sreg  function  unsop,
                dreg  COLON  EQUALS  sreg  shift  NUM,
                dreg  COLON  EQUALS  INPUT  variable,
                B  COLON  EQUALS  TRUE,
                B  COLON  EQUALS  FALSE;


codelist    =   inst,
                inst  SEMI  codelist;


elsepart    =   void,
                ELSE  codelist;


loopst      =   WHILE  condlist  DO  codelist  OD,
                REPEAT  codelist  UNTIL  condlist,
                BREAK,                      \ legal only in WHILE/REPEAT \
                CONTINUE;                   \ ditto \
```

```
\*************************  CASE statements  ******************************\

casest     =  CASE  caseitem  caselist  ESAC,
              CASE  preface  caseitem  caselist  ESAC,
              CASE  sreg  signtype  setitem  setlist  ESAC,
              CASE  preface  sreg  signtype  setitem  setlist  ESAC;


preface    =  BEGIN  simplist  END,          \ only assignments & calls .. \
              OPEN   simplist  CLOSE;        \ .. allowed in prefaces \


simplist   =  simstat,
              simstat  SEMI  simplist;


caseitem   =  condition  COLON  inst;


caselist   =  void,                          \ no ELSE present \
              SEMI,                          \ ditto, spare semi \
              SEMI  caseitem  caselist,
              opsemi  ELSE  codelist;        \ default limb \


signtype   =  IN,                            \ default type SIGNED \
              UNSIGNED;


setitem    =  range  COLON  inst;


setlist    =  void,
              SEMI,
              SEMI  setitem  setlist,
              opsemi  ELSE  codelist;


range      =  exp,
              exp  DOT  DOT  exp;
```

```
\****************************  Conditions  ********************************\

condlist    =   condition,
                preface  condition;          \ "preface" statements \

condition   =   B,
                NOT  B,
                TRUE,
                FALSE,
                cond,
                cond  AND  andlist,
                cond   OR  orlist;

andlist     =   cond,                        \ can't mix AND, OR \
                cond  AND  andlist;

orlist      =   cond,
                cond  OR  orlist;

cond        =   sreg  relop  exp,
                sreg  relop  ID  indxtail;

relop       =   LESS,
                GREATER  EQUALS,
                EQUALS,
                SLASH  EQUALS,               \ VIPER test instructions \
                LESS  EQUALS,
                GREATER,
                LT,
                GE;

opsemi      =   void,                        \ forgive spare semicolons! \
                SEMI;

sreg        =   DATAREG;                     \ one of A, X, Y \

dreg        =   DATAREG;
```

```
\************************  Operands and Functions  ****************************\

operand     =   exp,                          \ constant or scalar variable \
                ID  indxtail,                 \ vector reference \
                NOT  ID  indxtail;


unsop       =   unsexp,
                ID  indxtail;


indxtail    =   OSQ  offset  CSQ;


offset      =   unsexp,                       \ non-negative constant offset \
                indexreg,                     \ run-time index \
                indexreg  PLUS  unsexp,       \ ditto with constant offset \
                indexreg  MINUS  unsexp;


variable    =   ID,                           \ subset of "operand" \
                ID  indxtail;

indexreg    =   INDREG;                       \ one of X, Y \

function    =   PLUS,
                MINUS,
                ADD,        \ VIPER machine functions, not to be confused .. \
                SUB,        \ .. with operators in constant expressions \
                XOR,
                AND,
                NOR,
                AND  NOT;

shift       =   SLASH,
                GREATER  GREATER,
                TIMES,
                LESS  LESS;

\***************************  End of grammar  *******************************\
```

INTENTIONALLY BLANK

7 Using the VISTA translator

This description relates to VISTA Version 5 running on an IBM PC-compatible under MSDOS 3.3 or later.

The translator will print a summary of the operating instructions if invoked without arguments. Normally the name of the VISTA source file follows the program name; if no extension is given .VIS will be assumed e.g. `vista sourcefile`

VISTA can generate two output files, one containing diagnostics and information about the memory layout of the object program and the other containing the object program itself. The files normally have the same name as the source, and extensions (respectively) .ERR and .VOP but the names of the output files can be changed by an option.

Options are typed after the source file name, separated by a space. Every option begins with a letter (upper or lower case immaterial) and may include also a decimal number or a file name. Spaces may *not* appear within numbers or names, and a space (or end-of-line) *must* appear after a file name, but otherwise layout is unrestricted e.g.

```
vista sourcefile D5000 v p400b404 j2000o vopfile
```

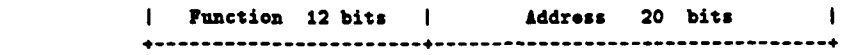
All options have default values which can be found by running the translator and inspecting *the statistics printed at the end. The options* available are as follows:

```
T          Do a trial translation only i.e. no .VOP file

V          Non-verbose output i.e. no comments in .VOP file.
           Verbose .VOP files are typically 3 times as big.

F 20       Allow for 20 source files i.e. up to 19 INCLUDE files.

D 12000    Allow for 12000 data declarations in the program
           (variables and constants).  Similarly P (procedure
           declarations) and R (region declarations).

J 5000     Allow for 5000 jumps - 3 each per IF, WHILE, REPEAT,
           CASE in the program.

C 200      Allow for 200 different anonymous constants i.e.
           constants appearing in instructions but not dec-
           lared explicitly with names.

B 1000     Allow 1000 blocks.  Blocks = procedures + 2

O name     Use "name.vop" and "name.err" for the two output files.
           Any extension typed will be ignored.  "O" is a letter.
```

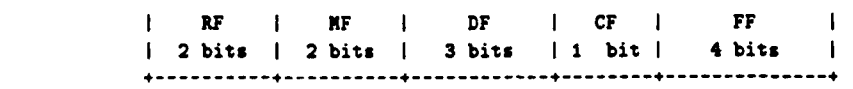
8 VIPER Machine Definition

The VIPER machine has 3 general purpose 32 bit registers (called A, X, Y), a program address counter (P), and a single bit Boolean register (B). Memory addresses occupy 20 bits, so only the least significant 20 bits of P are meaningful: loading a "1" into any of the top 12 bits will cause the machine to stop.

All instructions occupy 32 bits, divided into a 12 bit function code and a 20 bit address.



The function code is further divided into:



Most instructions are of the form D := R op M, where D and R are registers chosen from A X Y P. M is either a 20 bit literal constant or the contents of a 32 bit memory location. Memory addresses are limited to 20 bits, and the machine will stop if a computed address (MF = 2 or 3, see below) exceeds this limit.

RF: source register field

0	R	is contents of register A
1	R	" " " " X
2	R	" " " " Y
3	R	" " " " P after incrementing

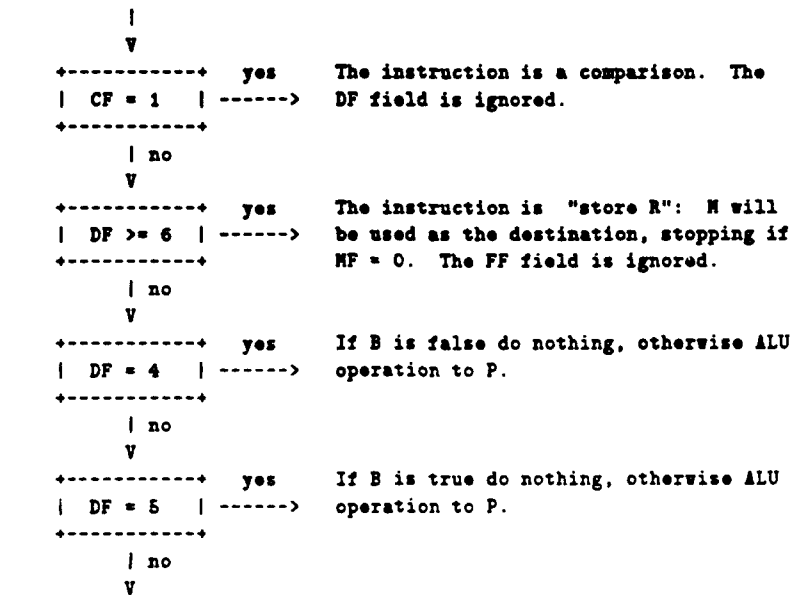
MF: memory address control field

0	M	is the address, i.e. 20 bit non-negative constant
1	M	is contents of address, memory or peripheral
2	M	is contents of (address+X) " " "
3	M	is contents of (address+Y) " " "

DF: destination control field

0	D	is the A register
1	D	" " X "
2	D	" " Y "
3	D	" " P "
4	D	" " P " if B, else do nothing
5	D	" " P " if NOT B, else do nothing
6	D	is M in peripheral space } see
7	D	is M in memory space } below

The sequence in which the CF, DF, and FF fields are inspected is important:



8.1 ALU operations

If CF = 0 and DF < 6, FF and possibly MF specify the ALU function:

```
FF = 0          D := NOT M  i.e. M operand complemented
     1          Y := P then P := M from memory space i.e. CALL
     2          D := M from peripheral space  }  equivalent
     3          D := M from memory space      }   if M = 0
     4          D := R + M,  B := carry
     5          D := R + M,  stop on overflow
     6          D := R - M,  B := borrow
     7          D := R - M,  stop on overflow
     8          D := R XOR M
     9          D := R AND M
    10          D := R NOR M
    11          D := R AND (NOT M)
```

```
12,  MF = 0     D := R/2, sign bit copied
     MF = 1     D := R >> 1 through B, i.e. D31 := B,
                            D0..30 := R1..31, B := R0
     MF = 2     D := R * 2, stop on overflow
     MF = 3     D := R << 1 through B, i.e. D0 := B,
                            D1..31 := R0..30, B := R31

13              spare instruction, stop
14              spare instruction, stop
15              spare instruction, stop
```

Note that if the destination is P (DF = 3, 4, or 5) only functions 1, 3, 5, and 7 are legal. Attempting to obey any other will cause the machine to stop. Function 1 operates only on P; if DF specifies any other destination register the machine will stop.

The operation D := R can be achieved by performing D := R + 0 or (for destinations A, X, Y) D := R AND NOT 0 which is faster. D := NOT R can be done as D := R NOR 0.

8.2 Comparisons

If CF = 1, FF specifies a comparison. Comparisons never change anything other than B, apart from the change in P implied by continuing to the next instruction, and never cause the machine to stop unless a memory address exceeds 20 bits (MF = 2 or 3). For comparisons the ALU function is forced to R - M, and the arithmetic unit behaves as if it had 33 bits with initial sign extension and no overflow detection, allowing signed or unsigned comparison of 32 bit values.

The new value of B is derived from the result of the subtraction as follows: bit32 is the conceptual extra (33rd) bit:

```
FF = 0      B := bit32                          (i.e. R < M)
     1      B := NOT bit32                           (R >= M)
     2      B := allzero                             (R = M)
     3      B := NOT allzero                         (R /= M)
     4      B := bit32 OR allzero                    (R <= M)
     5      B := NOT bit32 AND NOT allzero           (R > M)
     6      B := borrow                    (unsigned R < M)
     7      B := NOT borrow                (unsigned R >= M)
```

FF = 8 to 15: the set of operations is repeated, but if the result is 0 (i.e. FALSE) B is left unchanged.

9 VIPER Object Program (.VOP) Format

VIPER Object Program files are printable, and contain the information needed by a program loader, PROM programmer, or VIPER simulator. Only the information within the defined format is significant; the remainder of each line is undefined and can be filled with whatever material may be useful. The VISTA translator outputs the current loading address and a "reconstructed" version of each instruction, followed on some lines by a "marker" showing the beginning of loops, procedures etc.

The type of every line is indicated by the character in column 1, which is followed (in most cases) either by a string of characters, a hex number, or a sequence of decimal numbers. Some types of line have alternative decimal or hex forms. The defined formats must be adhered to rigidly.

In the following description, underscore indicates mandatory space characters. d stands for a decimal digit (0..9) and h for a hex digit (0..9, A..F or a..f). All other characters stand for themselves. Every defined format must end with a "white space" character i.e. space, TAB, or end-of-line.

```
T any character string
D any character string
```

Title and Date of program. These are not mandatory but are generated by VISTA from (respectively) the PROGRAM line which begins the source file and the date derived from the host operating sytem.

```
@_$hhhhhhhh
@_ddddd_ddddd
```

Set the currrent loading address i.e the address starting at which subsequent data will be loaded into memory. In the decimal form the second 5 digit number represents the least significant 16 bits of the address (0..65535) and the first number represents the rest. In both forms the digits are in the usual order i.e. most significant first. Addresses up to 16rFFFFF are interpreted as in memory. Addresses in the range 16r100000 to 16r1FFFFF are treated by the VIPER simulator as in peripheral space.

All the digits must be present in either format.

```
C_$hhhhhhhh
C_ddd_ddd_ddd_ddd
```

Load the 32 bit value given into the location pointed to by the current loading address, add 1 to the address, and update the sumcheck (see below). In the decimal form the four numbers (0..255) represent the four bytes of the value, most significant first. All the digits must be present.

```
V_$hhhhhhhh
```

Verify the current sumcheck against the hex value given, and clear the sumcheck. The sumcheck is the 32 bit residue of the sum of every 32 bit value loaded plus its address, since the sumcheck was last cleared. The sumcheck is zero initially. All the digits must be present.

```
E
```

End of .VOP file.

The remaining formats are relevant only to a simulator or a prototyping system which contains a host computer of some kind. PROM programmers ignore them.

```
A_$hhhhhhhh
A_ddd_ddd_ddd_ddd
```

Format as for C. Load the VIPER `A` register with the 32 bit value given. Similar formats exist for the `X`, `Y`, and `P` (program address) registers.

```
B_TRUE
B_FALSE
S_TRUE
S_FALSE
```

Set/clear the `B` flag or the internal `STOP` flag.

```
M_ddddddd_ddddddd
I_ddddddd_ddddddd
```

Define a memory or I/O region size, for memory allocation in the simulator. The first decimal number is the lowest address in the region, the second is the highest. In a VISTA output these correspond to the regions declared by the programmer, with the upper bounds as actually used rather than as declared. The bounds must be decimal, 0..1048575, but in these two formats leading zeroes may be omitted.

```
U any character string
```

The character string is the name of a file "used" by an `INCLUDE` statement. Simulators can use this information to improve diagnostics.

```
Q any character string
```

Corresponds to `DISPLAY` in the VISTA source. The character string may be interpreted by a simulator program or simply displayed as a diagnostic.

References

[Ker 87] Kershaw J: "The VIPER Microprocessor", RSRE Report 87014, 1987.

[Cur 84] Currie I F: "Orwellian Programming in Safety- Critical Systems", Conference on Systems Implementation Languages - Practice and Experience, University of Kent at Canterbury, 1984.

[RTP] MALPAS is supplied by Rex, Thompson and Partners Ltd, West Street, Farnham GU9 7EQ, UK.

[PVL] SPADE is supplied by Program Validation Ltd, 26 Queen's Terrace, Southampton SO1 1BQ, UK.

INTENTIONALLY BLANK

REPORT DOCUMENTATION PAGE

DRIC Reference Number (If known) .....

Overall security classification of sheet .....UNCLASSIFIED.....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the field concerned must be marked to indicate the classification eg (R), (C) or (S).

Originators Reference/Report No.	Month	Year
REPORT 91034	JANUARY	1992

Originators Name and Location
RSRE, St Andrews Road
Malvern, Worcs WR14 3PS

Monitoring Agency Name and Location

Title
THE VISTA STRUCTURED ASSEMBLER

Report Security Classification	Title Classification (U, R, C or S)
UNCLASSIFIED	U

Foreign Language Title (in the case of translations)

Conference Details

Agency Reference	Contract Number and Period

Project Number	Other References

Authors	Pagination and Ref
KERSHAW, J	31

Abstract

VISTA is a structured assembly language for the VIPER microprocessor chip. Though the syntax of VISTA and the appearance of VISTA programs are reminiscent of a high-level language, the actual statements are VIPER machine instructions. VISTA provides the clarity and much of the convenience of a true high-level language without the need for a complex and possibly untrustworthy compiling program.

This Report describes the VISTA language by means of an annotated example, and gives instructions for using the VISTA translator. A formal grammar of VISTA, and descriptions of VIPER and the VIPER Object Program format, are included.

Abstract Classification (U,R,C or S)
U

Descriptors

Distribution Statement (Enter any limitations on the distribution of the document)
UNLIMITED

INTENTIONALLY BLANK